

Design Specifications Document

Submitted By
Team 9: Uranus
James Kradjian
Jonathan Redus
Yil Verdeja

Date Submitted: February 28, 2018
Data Completed: February 14, 2018
Course Instructor: Loris Fichera
Lab Section: RBE 3001 C01

TABLE OF CONTENTS

LIST OF FIGURES	3
LIST OF TABLES	3
1 INTRODUCTION	4
1.1 The Robotic Manipulator	4
2 SYSTEM DESIGN	5
2.1 Overall Software Design	5
2.1.1 Firmware Programming	5
2.1.2 Matlab Programming	5
2.2 System Architecture	6
2.2.1 Servers	6
2.2.2 PID Tuning	7
2.2.3 Position Kinematics	8
2.2.4 Differential Kinematics	11
2.2.5 Motion Planning and Trajectory Generation	12
2.2.6 Live Plotting	13
2.2.7 Computer Vision	14
2.2.7.1 Camera Tracking	15
2.2.8 Object Weighing	15
2.2.9 Sorting	16
3 Organization	17
3.1 Refactoring	17

LIST OF FIGURES

Figure 1	Home Position of Robotic Manipulator	4
Figure 2	System Diagram	5
Figure 3	Status server and Calibration server operational diagrams	7
Figure 4	Reference Frame of Elbow Manipulator	8
Figure 5	Position Forward Kinematics Implementation	9
Figure 6	Full Drawing of the Inverse Kinematics	10
Figure 7	Error Cases inside the Inverse Kinematics Function	10
Figure 8	Numeric Inverse Kinematic Algorithm Flow Chart	12
Figure 9	CAD model of the robot arm	14
Figure 10	Image Processing Process	15
Figure 11	Example Sorting Configuration	17

LIST OF TABLES

Table 1	DH parameters of Elbow Manipulator	9
----------------	------------------------------------	---

1 INTRODUCTION

The goal of the final project is to build an automated robotic sorting system, where the robot will localize certain objects within its workspace, pick them up, classify them based on weight and appearance, and release them within a predefined area.

The objective of this document is to provide a deeper understanding of how the overall software is designed, and how the architecture of the software is structured to make the robot function as specified. This document will also specify what has been completed, what must be improved, and lastly what must be done to complete the project. The system design section will go in depth in describing how the robot will be capable of completing the goal of the project.

1.1 The Robotic Manipulator

The robot manipulator is a 3 degree of freedom (DoF) elbow manipulator with no prismatic joints. The base joint rotates about the Z axis, whilst the second and third joint rotate about the Y axis. This can be seen on Figure 1, which shows the robotic manipulator at home position.

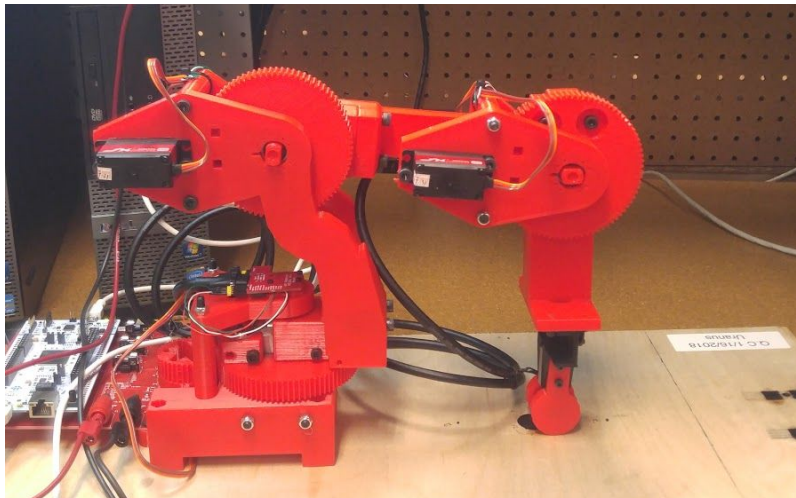


Figure 1. Home Position of Robotic Manipulator

When the robot is at home position, the end effector position should be at a position of zero in the X, Y and Z plane. This will be further discussed in section 2.2.1 which will show how this feature was implemented. The given link lengths of the robot were given as 0.135, 0.175, and 0.16928 meters for links one, two and three respectively, however they may be changed in order for more precise plotting for the stick model.

2 SYSTEM DESIGN

This section will discuss the layout of the code. The code is divided into two categories: firmware, which runs on a Nucleo board, and Matlab code, which runs on a PC.

2.1 Overall Software Design

The firmware and Matlab code communicate with one another to make the robot achieve tasks. The following is an overview of how the system will work:

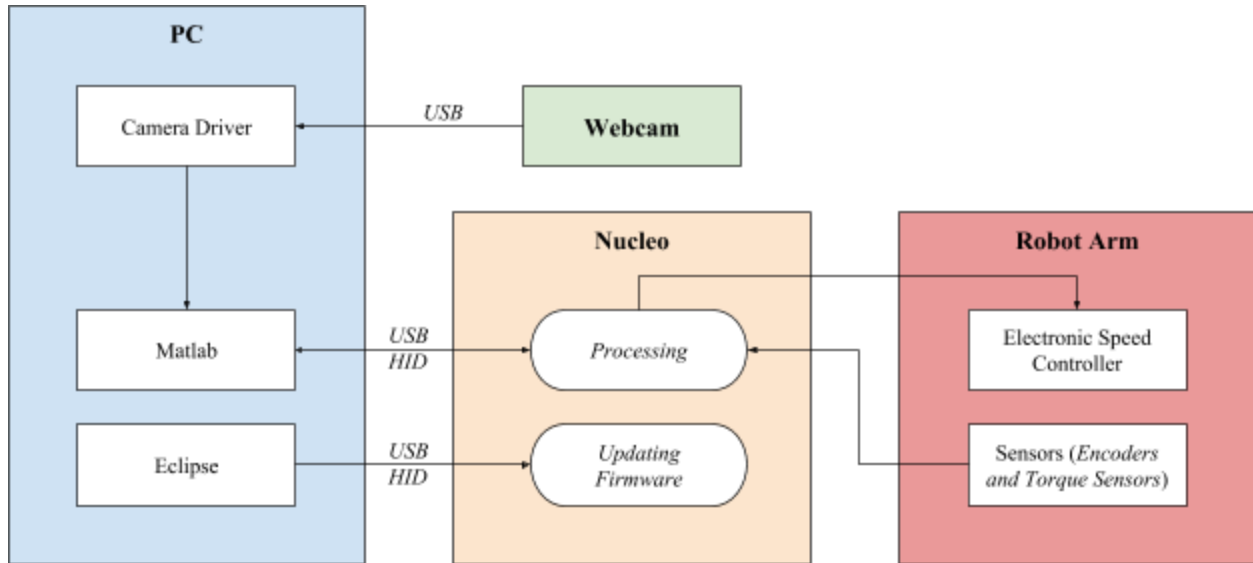


Figure 2. System Diagram

2.1.1 Firmware Programming

The Nucleo, the microcontroller running the firmware, has two interfaces to the host computer: a virtual serial port, to update the firmware, and a HID port, to communicate with the host computer. The virtual serial port is rarely used, since the information the servers send and receive is enough to complete the lab. The HID port operates with the Nucleo as a USB slave device. The host computer sends packets to the microcontroller, which are then interpreted in the firmware, and either a packet is sent back to the computer or the robot completes some action. Using this system, the host computer can move the robot to any position within the working area. This allows the host computer to cause the robot to pick up the objects.

2.1.2 Matlab Programming

Matlab on the host computer does most of the calculations, as it is on a mid-end Intel i5. The i5 is significantly faster than the Nucleo, since it is a high powered application processor. Section 2.2 goes into further detail on the Matlab code.

2.2 System Architecture

This section will discuss the subsystems of the code, including the theory behind the code as well as flowcharts and code snippets to show how the code works.

2.2.1 Servers

In order to communicate between the robot and the computer, several servers had to be implemented: the calibration server, the PID configuration server, the PID server, and the status server. All servers communicated with 16 byte words, where the first byte was the server ID for the specific server, so that all servers could be run at once.

The PID and PID configuration were created by a third party, and existed before the course began. The first server created for the project was the status server. It sends an empty packet from Matlab to the robot when the status command is called. The robot then reads the position, velocity, and force values from the hardware, and sends the data back in a packet of the form: Server ID; Joint 0 position; Joint 0 velocity; Joint 0 force; Joint 1 position; Joint 1 velocity; Joint 1 force; Joint 2 position; Joint 2 velocity; Joint 2 force; six empty values. The calibration server works whenever the calibrate command is called from Matlab. Matlab sends a packet with the position values of the arm. The firmware would then update the home position, by subtracting the values sent to it from the previous home values.

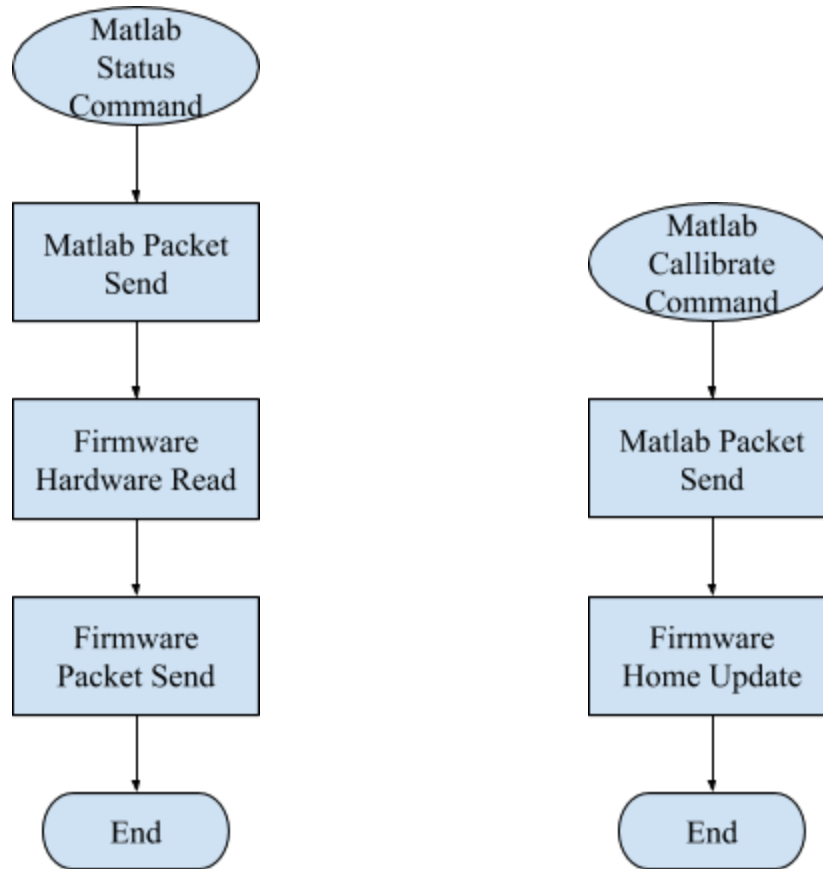


Figure 3. Status server and Calibration server operational diagrams

In this lab, the status server will have to be updated to average the values of the torque sensor. This is because the torque sensor is prone to inaccuracies. This will be accomplished by reading the value from the force sensor several times in the firmware when the status command is called, then averaging the readings. This will be done in the firmware and not the Matlab code because Matlab will take significantly longer to do call several status packets and take the average than using the firmware would.

2.2.2 PID Tuning

In previous labs the PID was tuned by looking at the motion of the robot alone. However, in this lab, external forces and torques have to be considered in order for the robot to have smooth transitions between different positions in the workspace.

To take the weights into account, either the firmware code on the PID controller has to change to compensate gravity and torque at the end effector, or seperate gains will be calculated for the different weights that might be in the system.

To get the most accurate results, the following steps will be followed to tune the PID:

1. Set all the gains to 0
2. Increase the Proportional Gain K_p until there is some overshoot (10-20% overshoot)
3. Increase the Integral Gain K_i until the waveform shows it is going to steady state
4. Increase the Derivative Gain K_d until oscillation is removed, and the step response is very stable
5. Tweak the gains if the desired response has not been achieved

In order to get better results, various motions will be tested and their position trajectories will be plotted. In that case, every time a gain for the PID control is changed, it can be seen throughout each movement how that change is affecting it.

2.2.3 Position Kinematics

In order to obtain the relationship between the position and joint angles, both forward and inverse kinematics had to be implemented. By drawing out a reference frame, and figuring out the DH parameters of the elbow manipulator (See Figure 4 and Table 1 respectively), given a set of joint angles the position of each joint could be found using forward kinematics.

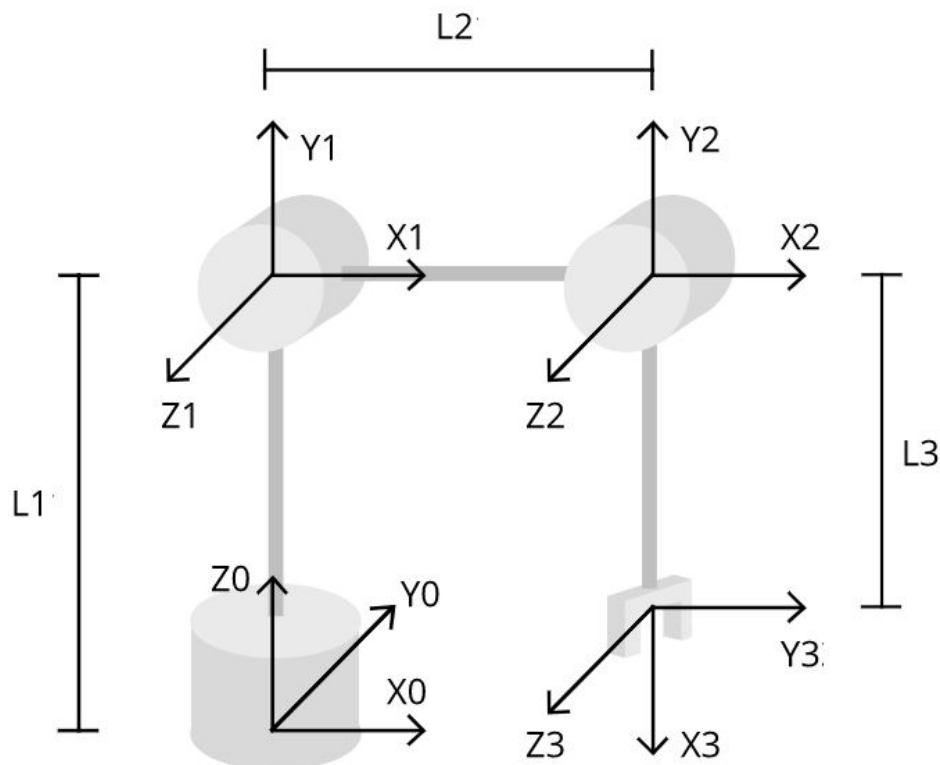


Figure 4. Reference Frame of Elbow Manipulator

Table 1. DH parameters of Elbow Manipulator

Links	θ	α	a	d
1	θ_1	90°	0	L1
2	θ_2	0	L2	0
3	$\theta_3 - 90^\circ$	0	L3	0

The math can be seen within the *fwkin3001* function below where transformation matrices are used to obtain the position of each joint in the workspace. The *getPosV* function gets the position vector of the transformation matrix which is the first three rows on the 4th column.

```
function [ pos ] = fwkin3001(q)
    % Returns the position of joints of a given set of joint angles

    %Length of Robot Arms
    L1 = 0.135; %[m]
    L2 = 0.175; %[m]
    L3 = 0.16928; %[m]

    % Gets the transformation matrices for all joints
    % tdh(theta, d, alpha, a)
    T0_1 = tdh(q(1), L1, sym(pi/2), 0);
    T1_2 = tdh(q(2), 0, 0, L2);
    T2_3 = tdh(q(3)-sym(pi/2), 0, 0, L3);

    % Calculates Transformation Matrices for remaining points from origin
    T0_2 = T0_1*T1_2;
    T0_3 = T0_1*T1_2*T2_3;

    % Retrieves the position values of all joints
    p1 = getPosV(T0_1);
    p2 = getPosV(T0_2);
    p3 = getPosV(T0_3);

    pos = double([p1;p2;p3]);

end
```

Figure 5. Position Forward Kinematics Implementation

However, to move the robot to a specific position in the workspace, the orientation has to be found. This is done through inverse position kinematics using a geometric approach. Looking at the *ikin3001* function and Figure 6, the joint angles are given by the following equations:

$$\theta_0 = \text{atan2}(P_Y, P_X)$$

$$\theta_2 = \arccos\left(-\frac{L_2^2 + L_3^2 - ((P_z - L_1)^2 + P_x^2 + P_y^2)}{2L_2L_3}\right)$$

There are two possibilities for the last angle θ_1 that depend on the angle of the second joint θ_2 .

$$\theta_1 = \arctan\left(\frac{P_z - L_1}{\sqrt{P_x^2 + P_y^2}}\right) + \arcsin\left(\frac{L_3 \sin(\theta_2)}{\sqrt{(P_z - L_1)^2 + P_x^2 + P_y^2}}\right) \text{ when } \theta_2 \geq 0 \text{ or}$$

$$\theta_1 = \arctan\left(\frac{P_z - L_1}{\sqrt{P_x^2 + P_y^2}}\right) - \arcsin\left(\frac{L_3 \sin(\theta_2)}{\sqrt{(P_z - L_1)^2 + P_x^2 + P_y^2}}\right) \text{ when } \theta_2 < 0$$

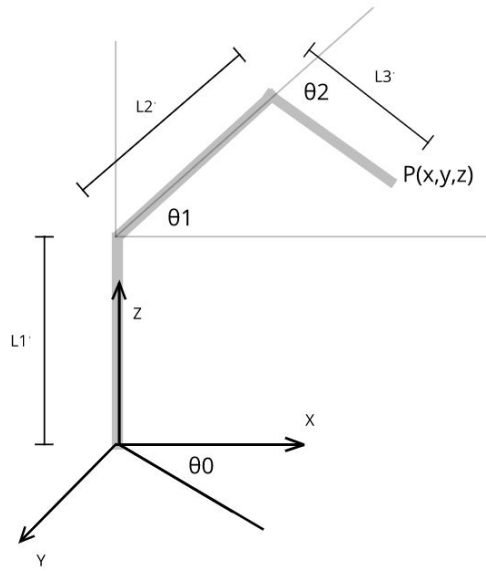


Figure 6. Full Drawing of the Inverse Kinematics

Another approach to solve for the inverse kinematics would have been to use the symbolic end effector position obtained from the transformation matrices and solve for the joint angles. However the geometric approach was chosen due to its simplicity. To make sure that the positions passed into the inverse kinematics function were within the workspace, error cases were included as seen in Figure 7. This ensured that the robot would not be sent to place it could not reach.

```
%error messages
if (p(3) > (L1 + L2 + L3) || p(3) < 0)
    error("Invalid Position: Z");
elseif (p(2) > (L2+L3) || p(2) < -(L2+L3))
    error("Invalid Position: Y");
elseif (p(1) > (L2+L3) || p(1) < -(L3))
    error("Invalid Position: X");
end
```

Figure 7. Error Cases inside the Inverse Kinematics Function

2.2.4 Differential Kinematics

There are several Matlab functions which utilize differential kinematics: *fwDiffKin*, *invDiffKin*, and *numInvKinAlg*. All of these depend on the jacobian, which is generated using the *jacob0* function.

The jacobian is a matrix which represents the relationship between the joint variables of the robot and the position and orientation of the end effector. It has the form:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \dots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

In order to calculate this in Matlab, the jacobian is broken into two parts: the position jacobian and the orientation jacobian. These are related to the jacobian as follows:

$$J(q) = [J_p(q); J_o(q)]$$

These can be represented by the following equations:

$$J_p(q) = [\dots \hat{\mathbf{a}}_i \times (p_e - p_i) \dots]$$

$$J_o(q) = [\dots \xi \hat{\mathbf{a}}_i \dots]$$

Where $\hat{\mathbf{a}}_i$ is the 3rd column of the rotation matrix from the transformation matrix of joint i , p_i is the position of joint i , and p_e is the position of the end effector, and ξ is 1, because this only needs to consider rotational joints. Thus, using the DH matrices derived in lab 2, a function to calculate the Jacobian based on the joint angles of the robot was created. For the base joint the $\hat{\mathbf{a}}_0$ is $[0;0;1]$ column vector, whilst the position p_0 is at $[0;0;0]$. For the second and third joint, transformation matrices T_0^1 and T_0^2 are used to determine $\hat{\mathbf{a}}_i$ and p_i . All of this was implemented in Matlab, to produce the function *jacob0*, which took the joint angles of the robot in, and returned the jacobian.

The first function which used the jacobian was *fwDiffKin*, which calculated the velocity vector of the end effector, given the joint angles and velocities of the robot, using forward differential kinematics. It did so by calculating the jacobian, by using the *jacob0* function, then multiplying the jacobian by the joint velocities.

Next, the *invDiffKin* function was made. This calculated the joint velocities, given the joint angles of the robot and velocity vector of the end effector, through the use of inverse differential

kinematics. It works by calculating the jacobian, using the *jacob0* function, then inverting the jacobian and multiplying it by the velocity vector of the end effector.

The final differential kinematics function was *numInvKinAlg*. This moved the robot to a certain position by using an inverse differential kinematics- based approach, where the difference between the desired position and the actual position, or approximately the position velocity, was multiplied by the inverse jacobian, which resulted in the change in joint angles to move toward the desired position. This would not get all the way there though, so the process was iterated until the actual position was close to the desired position. There were a few challenges posed by this system. The first was that desired position had to be checked to make sure that the robot could reach it. Otherwise, the results could be unpredictable. The other problem, unique to the specific application of the inverse kinematics based approach used in this lab, was that the calculated change in joint angle had to be divided by three. There is no theoretical explanation as to why this should work, but it was found that the system was unstable without this modification.

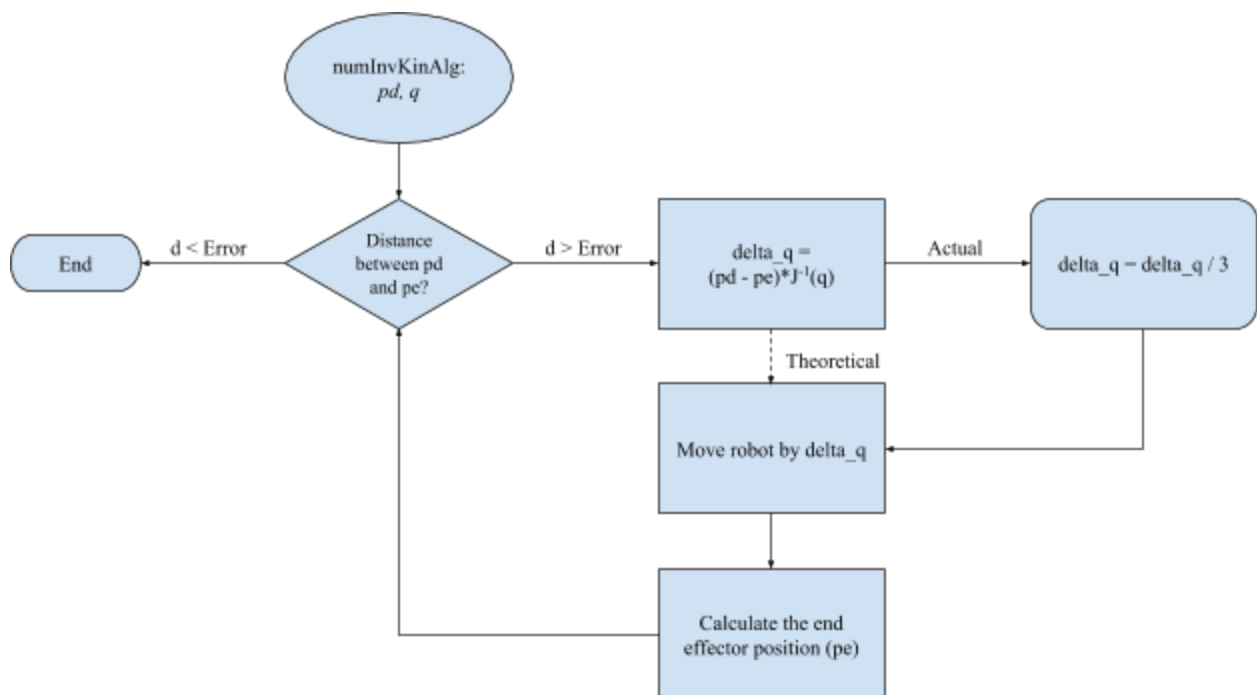


Figure 8. Numeric Inverse Kinematic Algorithm Flow Chart

2.2.5 Motion Planning and Trajectory Generation

Although the robot can easily be made from one point to another, there are two methods of trajectory calculation which result in smoother motion- cubic polynomial and quintic polynomial trajectory curve generation. These methods are very similar, except that the first uses a quintic polynomial and does not account for acceleration, while the second uses a quintic polynomial and accounts for acceleration.

The cubic polynomial trajectory solves the following equations to generate the appropriate constants, so that the trajectory can be calculated at any given point:

$$\begin{aligned} a_0 + a_1 * t_0 + a_2 * t_0^2 + a_3 * t_0^3 &= q_0 \\ a_1 + 2 * a_2 * t_0 + 3 * a_3 * t_0^2 &= v_0 \\ a_0 + a_1 * t_f + a_2 * t_f^2 + a_3 * t_f^3 &= q_f \\ a_1 + 2 * a_2 * t_f + 3 * a_3 * t_f^2 &= v_f \end{aligned}$$

The quintic polynomial trajectory solves the following equations to generate the appropriate constants, so that the trajectory can be calculated at any given point:

$$\begin{aligned} a_0 + a_1 * t_0 + a_2 * t_0^2 + a_3 * t_0^3 + a_4 * t_0^4 + a_5 * t_0^5 &= q_0 \\ a_1 + 2 * a_2 * t_0 + 3 * a_3 * t_0^2 + 4 * a_4 * t_0^3 + 5 * a_5 * t_0^4 &= v_0 \\ a_2 + 6 * a_3 * t_0 + 12 * a_4 * t_0^2 + 20 * a_5 * t_0^3 &= a_0 \\ a_0 + a_1 * t_f + a_2 * t_f^2 + a_3 * t_f^3 + a_4 * t_f^4 + a_5 * t_f^5 &= q_f \\ a_1 + 2 * a_2 * t_f + 3 * a_3 * t_f^2 + 4 * a_4 * t_f^3 + 5 * a_5 * t_f^4 &= v_f \\ a_2 + 6 * a_3 * t_f + 12 * a_4 * t_f^2 + 20 * a_5 * t_f^3 &= a_f \end{aligned}$$

2.2.6 Live Plotting

The position of the robot will be plotted in real time, in a 3D dimensional space. This will use basic forward kinematics to calculate the position of each joint. Furthermore, each linkage will be modeled using CAD software, resulting in something similar to the model in Figure 9.

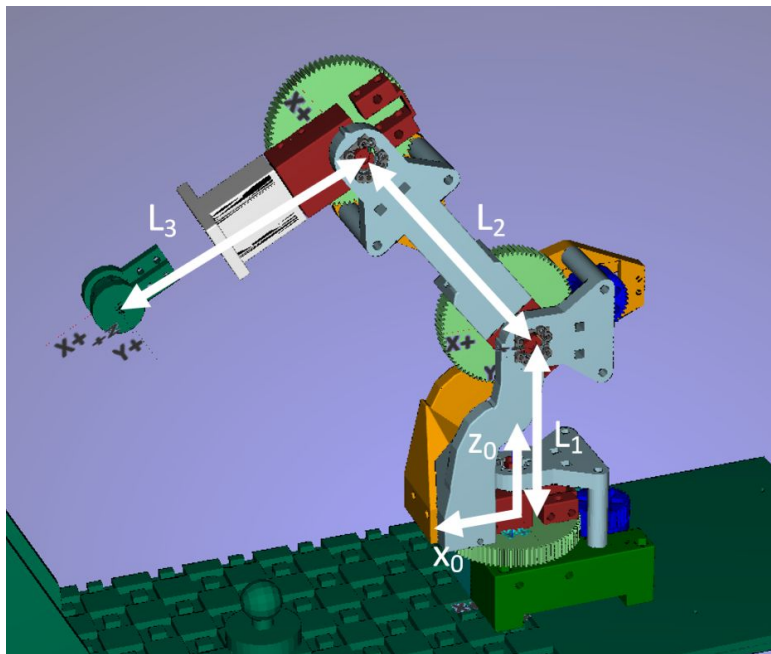


Figure 9. CAD model of the robot arm

2.2.7 Computer Vision

In the lab, computer vision will be implemented, in order to identify the position and color of objects within the workspace of the robot. To do this, an image, taken with the provided webcam, will be segmented by the color of the objects, and then the centroids will be identified. To segment the image properly, it will be segmented three times- one for each of the possible colors of the objects- using color filtering. The specific color space will vary based on what produces the most distinct section for the color being looked for. Once the image has been segmented, it will be processed, to determine the centroids of all objects present. This will allow for the location of each object to be found, using the mapping from pixel space to robot workspace generated using the *calibrate_camera* function.

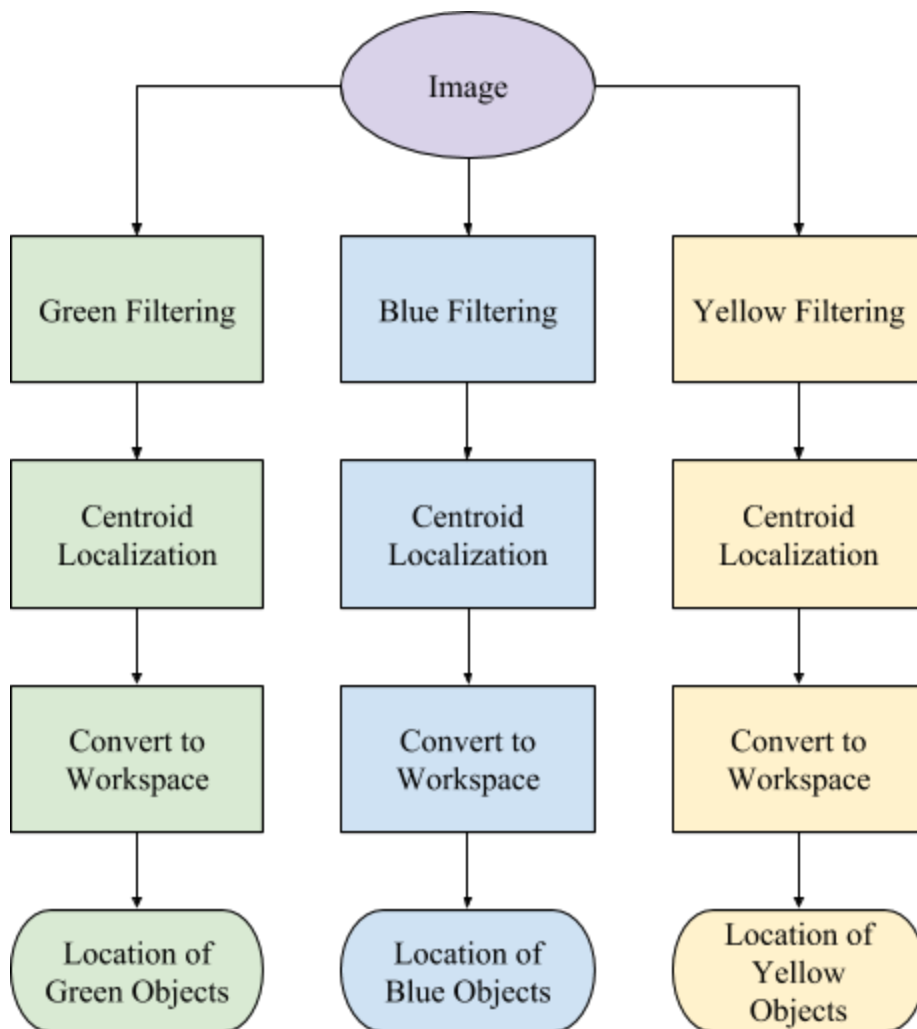


Figure 10. Image Processing Process

2.2.7.1 Camera Tracking

Once basic computer vision is created, it will be used to track the position of the object in real time, so that if the object moves, the arm will follow it until it catches up. This will require a way to specifically identify each object of interest in the work space, so that it consistently follows one object. It will also require repeated calls of the computer vision function, which means that it must be quick, or it will cause detrimental delays in the overall system.

2.2.8 Object Weighing

Forces sensing will also be implemented, in order to determine the difference between heavy and light objects. To do this, the torque readings, transmitted to Matlab via the status command, will be converted into force, compared to a threshold, and evaluated. The conversion from torque to force can be calculated using the jacobian:

$$\tau^T = J(q)^T F_t$$

Where τ^T is the transpose of the torque vector, $J(q)^T$ is the transpose of the jacobian, and F_t is the force vector. Thus, the force can be calculated using:

$$F_t = J(q)^{-T} * \tau^T$$

Where $J(q)^{-T}$ is the inverse of the transpose of the jacobian. A function will be made which will take the ADC values from the torque sensor, convert it into Newton meters, then multiply the torques by the transpose of the jacobian.

2.2.9 Sorting

Considering that the objects will differ both by color and by weight, once the system specifies an object, it should identify its color, determine where it is, pick it up and weight it. All this will lead to either two solutions:

1. Sort the object based on its color and weight: Outside of the camera workspace, but within the robots workspace, the objects will be placed accordingly as envisioned in Figure 11.
2. Make a unique action for each object type: This can either be an action for a specific color and an action for a specific weight, or an action for both a weight and color that is uniquely different from the rest.

Assuming that the objects are not within the camera workspace, an example of how sorting could occur depending on both the objects color and weight is shown in Figure 11 below. As seen, there are yellow, green and blue objects with different weights specified by the color black (plastic) and gold (copper). Figure 11 is an example of a sorting configuration the robot may follow.

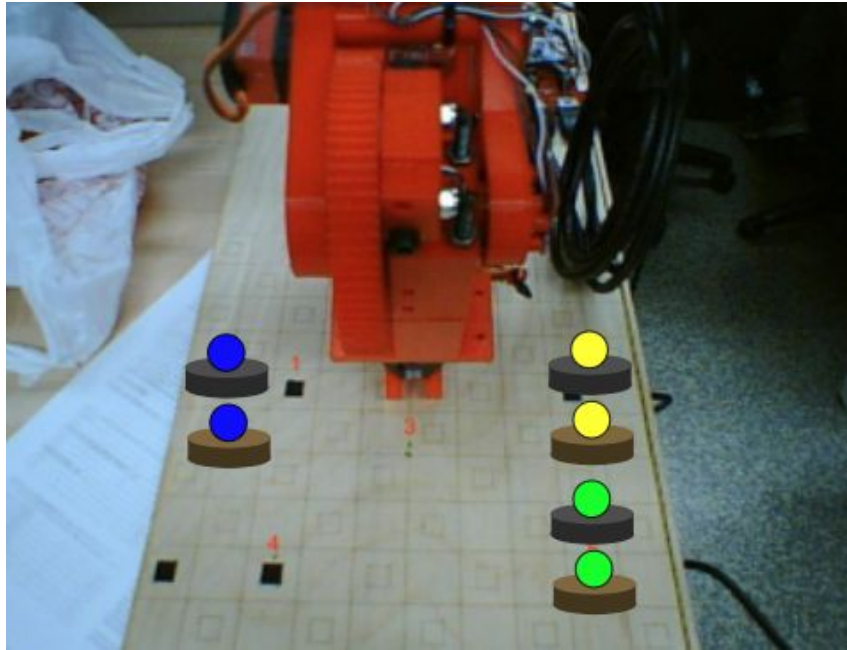


Figure 11. Example Sorting Configuration

3 Organization

Currently, the Matlab code is separated into three parts: the scripts, the functions, and the CSV files. The scripts are demonstrations of the functions, and while they are what runs the process that the robot goes through, they are not the majority of the work. The functions are the majority of the work. Having most of the work contained within the functions means that the code is much more reusable than if significant portions of code were written in the scripts. It also means that the code is generally more organized than the long blocks which sometimes come up when writing a single script to achieve a task. The final category is the CSV files. While these are not code, nor did anyone explicitly write these, they are important, because they have all the data in a form which can easily be processed by Matlab or outside programs. This means it is easy to analyze the data.

3.1 Refactoring

Most of the code written was to make a function work as expected. If it proved functionality, it was enough to go to the next part. However, code that works does not qualify for optimal code. A big problem in the current system is slow live-plotting. Removing repeated and unnecessary code may result in faster processing speeds for live plotting and camera tracking, thus making the code more optimal in its functionality, readability and speed.